**gBase**

**class library**

**Table of contents**

**User's reference 4**

Before you start 4

What's inside the box 4

Installing **gBase** 5

Tables 5

Views 8

Adding rows 10

Retrieving rows 10

Updating rows 11

Ready to fly on your own! 12

**Technical reference 13**

**class Table 13**

**Methods 13**

Add 13

Blank

Close 14

Create 14

Delete 16

Deleted 16

Destroy 17

Exists 18

Find 19

First 20

Jump 21

Last 22

Lock 23

Next 24

OnLoad 25

OnSave 26

Open 27

Pack 29

RelativeRow 30

Previous 31

Recall 31

Rows 32

Unlock 33

WriteAndUnlock 34

**Variables 35**

TableFileName 35

Row 36

**Static Methods 37**

Fix 37

GetPath 38

ReadFix 38

SetPath 39

WriteFix 40

**User's reference**

**Before you start**

* + This software was designed for minimum footprint and maximum efficiency; features that block these goals were not included and will not be included in the future.

* + This software was designed to set you up and running in minimum time, so the simplest data base model was chosen.
  + This software was designed for small to medium size projects; it is not meant to be used on large scale projects, and must not be used on critical applications or highly intensive transaction systems.
  + This software is designed to run on generic, stand alone hardware. It is not prepared to take advantage of special features found on dedicated hardware, and it is not prepared to run on distributed infrastructure.
  + This software was designed to integrate quickly, easily and directly to any **C#** project; this document does not provide information about how to access the software from other languages, so if you want to do so, you must find out the right way by yourself.
  + This document assumes you have at least basic knowledge of **C#**, and good understanding of classes and objects, if you don't have that background, please take the time to learn before you continue.

**What's inside the box**

The software is distributed in a **ZIP** file which contains the following two items:

* + The software itself which is contained on a **DLL** named **gBase.DLL**.
  + The user's manual contained in a file named **gBase.PDF**

**Installing gBase**

After unziping the files, you just need to drag and drop the **gBase.DLL** file to your **Assets** folder inside **Unity's** window. If you are not using **Unity**, just include the file in your project references. The exact procedure depends on the software you are using to build your project/solution, so please refer to your compiler’s user's manual if you don’t know how to do it.

Once you have included **gBase** on you project, please include the namespace **gBase** on any source file where you want to use it.

using gBase;

And voila, that's it, no additional steps are required.

**Tables**

A database is collection of tables; each table contains a variable number of rows containing a fixed set of columns. The best way to represent that idea in **C#** is to define a class composed by a set of data members that correspond to each column of the table, thus the class represents the columns, while the instances of such class represent the rows. The base class for any **gBase** table is: **Table**, which is declared as **abstract**, so it can only be used as base class, never directly.

The class that defines a table is known as the *structure of the table*, before you can create or open a table you must first define its structure, or in other words, declare a subclass of **Table**, containing at least one data member, both the class and the data members must be declared public to make them easily accessible.

public class User : Table{

public string email;

public string password;

public string name;

public string lastname;

}

Once the table has a defined structure, you must provide two helper methods, one to write the data members to disk (**OnSave**), and other one to read them from disk (**OnLoad**). The ***OnLoad*** method is called every time a row is read from the data file, a **BinaryReader** is provided as a mean for you to read the file buffer and populate your data members, you can safely use any of the **BinaryReader.Read** methods except for **ReadString** which must never be used. To read any string you must use **Table**'s static method **ReadFix**.

The ***OnSave*** method is called every time a *row* needs to be written to disk, a **BinaryWriter** is provided as a mean for you to write your data members to the file, you can use any of the overloaded **Write** methods except for strings, whenever you need write a string you must use **Table**'s static method **WriteFix** otherwise you may corrupt the file, this is especially true if you write a string that contains special characters.

Not all data members need to written to disk, however ever member written by ***OnSave*** must read by ***OnLoad*** in exactly in the same sequence, otherwise you won't be able to get any useful information out of your table.

Special care must be taken in case of the strings; you must be sure the size of the string being read is exactly the same as the string that was written, so the size parameter specified for **WriteFix** at the moment of writing the string must be the same passed to **ReadFix** at the moment of reading, as shown in the example below:

public override void OnLoad(BinaryReader rd){

email = ReadFix(rd,60).Trim();

password = ReadFix(rd,15).Trim();

name = ReadFix(rd,40).Trim();

lastname = ReadFix(rd,40).Trim();

}

public override void OnSave(BinaryWriter wr){

WriteFix(wr,email,60);

WriteFix(wr,password,15);

WriteFix(wr,name,40);

WriteFix(wr,lastname,40);

}

If you fail to enforce fixed size strings, **gBase** may throw an exception while reading or writing or even refuse to open the table at all for a second time, so please be very careful with this point.

You can include any of the built-in **C#** data types inside your table. If you wish to save and retrieve data types that are not native to **C#** you must provide the appropriated code to read and write them, as in the next example:

public class Row {

public string Name;

public float Weight;

}

public class Page : Table {

public DateTime Date;

public ushort Rows;

public Row[] List;

public override void OnLoad(BinaryReader rd) {

Date = DateTime.FromBinary(rd.ReadInt64());

Rows = rd.ReadUInt16();

for(ushort i=0;i<10;i++) {

List[i].Name = ReadFix(60).Trim();

List[i].Weight = rd.ReadSingle();

}

}

public override void OnSave(BinaryWriter wr) {

wr.Write(Date.ToBinary());

wr.Write(Rows);

for(ushort i=0;i<10;i++) {

WriteFix(wr,List[i].Name,60);

wr.Write(List[i].Weight);

}

}

Once you defined the structure of the table and the helper methods, you are ready to create the table file by calling non-static method **Create**. Notice you must provide the file name before calling **Create**, this allows you to have two or more tables with the same structure on different files.

User usr = new User();

Usr.TableFileName = "users";

usr.Create();

There are two possible ways to assign the file name of a table, if you are going to have multiple tables with the same structure you must specify the file name outside the class, as in the previous example. But if you only need one file per table you also have the option to add a constructor to your class in which you assign **TableFileName** with the desired value as follows:

public User(){

TableFileName = "users";

}

The file is created in the *default path* which is the same folder that contains the executable file, if you want your database files to reside in a different folder please call static method **SetPath** with the appropriated path name at the beginning of your program.

If you are using **gBase** on a **Unity** project it is highly recommended that you set the default path to: **Application.persistentDataPath** before attempting any database operation especially if you are aiming to deploy on mobile devices, otherwise your scripts will crash and you won't have any visual clue about what's going on.

Table.SetPath(Application.persistentDataPath);

**Create** limits itself to create the file, it does not open the file. To open an existing table, call the non-static method **Open**. Observe that you must provide the file name before calling **Open**; the same techniques used to assign the file name to create the table apply for opening it.

User usr = new User();

usr.TableFileName = "users";

if(!usr.Exists()) usr.Create();

usr.Open();

**Views**

A *view* is an internal object that uses delegate method **mpIndexKey** to bind each row in the table with a key; keys allow the *view* to sort the table in any particular order you want, but also to quickly locate any row included on the *view*. Since keys are always strings, an empty string represents an empty key, empty keys are discarded form the view, so the **mpIndexKey** method can also be used to filter the table.

public delegate string mpIndexKey(Table obj);

When you open the table providing no parameters, a *physical view* is bound to the table. A *physical view* allows you to reach all the rows physically stored in the table in the exact sequence in which the rows were added to the table, but most of the times this isn’t useful.

If you wish to access the table in an ordered way you must provide an **mpIndexKey** method to generate a key for every row in the table, so the *view* can sort the rows in *key* order. For example, if you wish your table to be classified by last name and then by name, you must provide an **mpIndexKey** method to generate the proper keys, as shown below:

public static string ByLastName(Table row){

User usr = (User) row;

return Fix(usr.lastname,40)+Fix(usr.name,40);

}

Notice the call to method **Fix** on the example above, this ensures all the keys generated are exactly the same length, which guaranties they can be found later on.

You can pass this method to **Open**, so it can bind a *sorted view* to the table. The *view* creation process will perform a sequential scan of the table, from beginning to end in order to generate and index file that can contain all the bounds between the rows and their respective keys. If you provide no more parameters the *view* will be treated as temporary, so the index file will be destroyed as soon as you **Close** the *view*. If you additionally specify a file name, the sort will be treated as permanent and the index file will be kept after the *view* is closed, so next time it is requested, it will be just open and used as it is, instead of generated all over again from scratch, speeding up the process.

A table can have as many *views* as necessary, temporary *views* can be created and destroyed at any moment as there’s no performance gain by creating or destroying them at specific moments. However permanent *views* must be open in block before any update occurs and also closed in block after all updates have been done, otherwise the system will have to rebuild the index file every time the *view* is requested, which defeats the goal of a permanent *view*.

public static string ByEmail(Table row){

User usr = (User) row;

if(usr.lastname=="Smith") return ""; // Hide it

return Fix(usr.email,60);

}

.

.

.

User usr = new User();

usr.TableFileName = "users";

User use = new User();

use.TableFileName = "users";

usr.Open(User.ByLastName); // Temporary view.

use.Open(User.ByEmail,"user-email"); // Permanent view.

As you can see in the previous example, you can have two or more *views* simultaneously open on the same table; this allows you to have several sorts and filters available, so you can choose the one that best suits your needs for any particular task. The first **Open** binds ***usr*** to a *view* that sorts the table by ***lastname*** and then by ***name*** and includes all the rows on the table. The second **Open** binds ***use*** to a *view* that sorts the table by ***email*** and filters out all the rows where **lastname=="Smith"**, so while ***usr*** will be able to reach all the rows in the table, ***use*** won't be able to see any row where **lastname=="Smith"**.

You can define any key you want as long as you make sure it generates a fixed size string. The maximum key size is 65535 characters but the longer the key the bigger the index page and thus the more time it will take to read it from disk, so we recommend you to keep it as small as possible, 300 characters have enough room to store the most commonly used data, such as name, address, etc., so we recommend you to keep your keys below that limit whenever possible.

**Adding rows**

The main goal of a table is to store data, an empty table is in most cases worthless, so probably the first thing you want to do is to add some rows. A table is ready for update immediately after it’s open. To add new rows, all you need to do is to populate your data members and then call non-static method **Add**. For example:

usr.Blank();

usr.email = "smithjohn@bigcompany.com";

usr.password = "can't touch this";

usr.name = "John";

usr.lastname = "Smith";

usr.Add();

Any open *view* is updated after the row is added, due to this, the new row may become invisible to the *view* that created it, when this is the case, the *view pointer* becomes invalid and reset to row zero, and the *data members* are blanked. If this happens, please don't panic, the new row is safe on disk, your view is just unable to see it, so unless your code is prepared to deal with this situation, whenever possible use an unfiltered view when updating a table.

**Retrieving rows**

When the table is open, the data members of the class are filled with the data contained on the first row of the *view*, or blanked if the table is empty. From that point you have various ways to proceed. You can process the rows sequentially by placing a call to **Next** inside a **while** loop, and make a throughout scan of the table, as in the following example:

while(usr.Row!=0){

.

.

.

usr.Next();

}

In this example, **Row** will become **0** as soon as all the rows have been processed, while method **Next** will populate the **usr**’s data members with the data contained in the following row of the *view* on each iteration. Whenever you make a sequential scan and there is a chance the current row is not the first, you should call **First** before entering the loop.

You can also process the rows sequentially in reverse order by using the following code:

usr.Last();

while (usr.Row!=0) {

.

.

.

usr.Previous();

}

In the previous examples the rows are accessed one by one, but you can also **Jump** any number of rows, either in the positive (towards the bottom) or negative (towards the top) direction, jump to a specific row, or find rows by key. If your table is bound to a *sorted view* and you know the key of a specific row you can use **Find** to quickly retrieve that row.

if(usr.Find(Table.Fix("Smith,40)+Table.Fix(John",40))){

.

.

.  
 }

When successful, **Find** points the *view* to the found row, populates the class data members with the data contained on it and finally returns **true**. If the key is not found, the *view* is pointed to row zero, the class data members are blanked, and **Find** returns **false**. When the *view* is pointing to row zero, any further call **Next**, **Previous**, and **Jump** simply return without doing anything. To get out of row zero you can call, **First**, **Last**, **Find** or **Add**, or you can set **Row** to a visible row number but if the *view* is empty, the only way to get out of row zero is by adding a row that is visible for that view.

**Updating rows**

The update process is quite simple, the steps are the same as in any other database management tool, you first locate the row you want to update, either directly through **Find** or indirectly by sequential scanning, then you **Lock** the row to prevent other threads form gaining simultaneous access to it, update the required data members, write the row back to disk and finally release the lock. See the example below.

if(usr.Find(key)){

usr.Lock();

usr.email = newEml;

usr.password = newPwd;

usr.WriteAndUnlock();

}

Please remember that **Unlock** does not perform any update operation on the table, it just unlocks the row by discarding any changes. To have the row actually written to disk you must call **WriteAndUnlock**.

The write process includes the update of any open *view*, if as result of this update the row being updated becomes invisible to the *view* that updated it, the *view pointer* becomes invalid and reset to row zero, and the class data members are blanked.

**Ready to fly on your own!**

Congratulations!!!

If you were able to understand the concepts presented in the previous pages, then you are able to start exploiting **gBase**.

Good luck and let us know what are you using **gBase** for.

**Technical Reference**

**class Table**

**Namespace:** gBase

**Assembly:** gBase.DLL

**Syntax:**

abstract public class Table;

**Methods**

**Add**

**Goal:**

To add a new row to the table.

**Syntax:**

public void Add();

**Remarks:**

Adds a new row to the table and then updates any open *view*, if the new row becomes invisible to the *view* that created it, such *view* is reset to row zero and **this** is blanked.

Notice that due to the fact the new row was not previously know by other threads, there is no need for thread synchronization (**Lock**/**Unlock**).

**See also:**

**Lock, Unlock, WriteAndUnlock**

**Blank**

**Goal:**

Clears all data members.

**Syntax:**

public void Blank();

**Remarks:**

Often rows are added in response to a failed search operation, if that’s the case, the row pointer should be at the end of file which causes the data members to be automatically cleared, however if you are adding records when the row pointer is not at the end of file, you should explicitly call this function in order to avoid introducing garbage in the new rows.

**See also:**

**Lock, Unlock, WriteAndUnlock**

**Close**

**Goal:**

To destroys the current *view*, and close the table if necessary.

**Syntax:**

public void Close();

**Remarks:**

Destroys the *view* bound to the table, if the *current view* is the only one open for that table, the table is closed at the same time as the *view* is destroyed. If there is a physical deletion scheduled, it is carried out just before the table file is closed. If a thread tries to open a table while there is a packing in progress, such thread is suspended until the packing process has finished. To close *views* explicitly is necessary to keep the index files updated and avoid rebuilding them every time the table is open.

**See also:**

**Create, Destroy, Exists, Open, Pack**

**Create**

**Goal:**

To create a new table.

**Syntax:**

public void Create();

**Remarks:**

Creates a new table, the name of the file is assumed to be stored on **TableFileName** and then the suffix ".Table" is added to the file name. The file is created at the default folder which is the same where the executable file resides but a call to static method **SetPath** can change it to any other you want. To know where the default path is pointing to, use static method **GetPath**.

Once the file has been created a call to ***OnSave*** is issued in order to compute the number of bytes required to store the row, this value is saved to disk so **Open** can validate it.

**Create** limits itself to create the table file, it does not open the table, nor make it ready for any use, a call to **Open** is required to get the table actually open.

An exception is thrown if the table is already open or if a file with the same name exists on the default path.

If you are using **gBase** on a **Unity** project it is highly recommended that you set the default path to **Application.persistentDataPath** before any database operation takes place.

**See also:**

**Close, Destroy, Exists, GetPath, Open, SetPath**

**Delete**

**Goal:**

To flag a row for physical deletion.

**Syntax:**

public void Delete();

**Remarks:**

Flags the row pointed by the *view* as deleted. No action is carried out if the row was not previously locked. To make the change permanent, please call **WriteAndUnlock**.

Delete does not remove the row from the table; it just flags it for deletion. The row may remain on the table indefinitely until a **Pack** is scheduled.

**See also:**

**Deleted, Lock, Pack, Recall, Unlock, WriteAndUnlock**

**Deleted**

**Goal:**

To report if a row is flagged for deletion.

**Syntax:**

public bool Deleted();

**Remarks:**

Reports if the row pointed by the *view* is flagged for deletion, when the *view* is pointing to row zero **Deleted** returns always **false**.

**See also:**

**Delete, Recall**

**Destroy**

**Goal:**

To remove a table file from disk.

**Syntax:**

public void Destroy();

**Remarks:**

Attempts to delete the table. The file name is assumed to be contained on **TableFileName**, to have extension ".Table" and to exist at the default folder, which if not changed, is the same where the executable file resides. A call to static method **SetPath** can change the default path if necessary. To know what's the default use static method **GetPath**.

If you are using **gBase** on a **Unity** project it is highly recommended that you set the default path to **Application.persistentDataPath** before any database operation takes place.

The table must be closed to get a successful deletion. An exception is thrown if the table is open.

**See also:**

**Close, Create, Exists, GetPath, Open, SetPath**

**Exists**

**Goal:**

To find out if a table exists.

**Syntax:**

public bool Exists();

**Remarks:**

Reports if the table specified in **TableFileName** exists. The file is assumed to have extension ".Table" and to exist at the default folder, which if not changed, is the same where the executable file resides. A call to static method **SetPath** can change the default path if necessary. To know what's the default use static method **GetPath**.

If you are using **gBase** on a **Unity** project it is highly recommended that you set the default path to **Application.persistentDataPath** before any database operation takes place.

**See also:**

**Close, Create, Destroy, GetPath Open, SetPath**

**Find**

**Goal:**

To search a row by key.

**Syntax:**

public bool Find(string key);

public bool Find(string key,bool soft);

**Remarks:**

Reports if the specified ***key*** exists in the current *view*, if the ***key*** is found, the *view* is pointed to the corresponding row and the row is loaded into the data members of **this**. If the ***key*** is not found, the *view* is pointed to row zero, and the data members are blanked.

A successful search requires the length of the search ***key*** to be the same size as the keys generated by the **mpIndexKey** method used to sort the table, if this condition is not fulfilled **Find** won't be able to find any ***key*** you provide, unless you specify a soft search.

An exception is thrown if thetable is not bound to a *sorted view* (no **mpIndexKey** method was provided when the table was **Open**).

If parameter ***soft*** is specified and it is **true**, a soft search is performed, so in case the exact ***key*** is not found, the view will be pointed to the row that's bound to closest greater key, so the view will only be pointed to row zero if no greater key exists, this is useful when looking for partial keys or groups of keys, for example, if you had *view* that sorted a table by last name and name, you could provide to **Find** with only the last name part of the key and specify **true** in parameter ***soft***, that way **Find** will point your *view* to the first person with that last name on your table.

**See also:**

**First, Jump, Last, Next, Previous, Row**

**First**

**Goal:**

To point the view to the first row.

**Syntax:**

public void First();

**Remarks:**

If the *view* is not empty, **First** points the *view* to its first row and loads the row into **this**. If the *view* is empty, the *view* is pointed to row zero and **this** is blanked.

**See also:**

**Find, Jump, Last, Next, Previous, Row, Rows**

**Jump**

**Goal:**

To jump a specific number of rows.

**Syntax:**

public void Jump(long rows);

**Remarks:**

Uses the row pointed by the *current view* as starting point to jump the number of rows specified in ***rows***. If ***rows*** is positive the *view pointer* moves towards the last row (bottom), if ***rows*** is negative the *view pointer* moves towards the first row (top). Rows are jumped according to the order defined by the *view*. If the jump ends on a valid row, such row is loaded into **this**, otherwise the view is reset to row zero and **this** is blanked.

**Jump** requires a valid row as starting point for the jump, so when the *view* points to row zero no jump is possible, thus in this case **Jump** limits itself to return without doing anything.

**See also:**

**Find, First, Last, Next, Previous, Row, Rows**

**Last**

**Goal:**

To points the view to the last row.

**Syntax:**

public void Last();

**Remarks:**

If the *view* is not empty, **Last** points the *view* to its last row and loads that row into **this**. If the view is empty, the *view* is pointed to row zero and **this** is blanked.

**See also:**

**First, Find, Jump, Next, Previous, Row, Rows**

**Lock**

**Goal:**

To locks a row.

**Syntax:**

public void Lock();

**Remarks:**

Locks the row pointed by the *view* to keep other threads from writing it simultaneously. A thread can lock a row more than once, however when this happens, the lock is only released after the row was unlocked the same number of times it was locked. If a thread tries to lock a row which was previously locked by another thread, its execution will be suspended until the first thread releases the lock, so be extremely careful with the sequence of events if you need to lock rows on two or more different tables, always make sure tables are unlocked in the same sequence they were locked otherwise you run the risk of a deadlock. Two or more threads can read the row while it is locked, but only the one that owns the lock can update it.

To prevent data loss **Lock** reads the row from disk again and discarding any previously made changes, this means that any changes made before **Lock** and after **WriteAndUnlock** are discarded. Only those changes made between these two functions will persist. To release the lock without making updates to the table please call **Unlock**.

**See also:**

**Add, Unlock, WriteAndUnlock**

**Next**

**Goal:**

To jump one row towards the bottom of the table (last row).

**Syntax:**

public void Next();

**Remarks:**

If the row pointed by the *view* is not the last one, the *view* jumps one row towards the last row, and the newly pointed row is loaded into **this**. If the *view* is pointing to the last row, the *view* is reset to row zero and **this** is blanked.

**Next** requires a valid row as starting point for the jump, when the *view* points to row zero no jump is possible, thus **Next** limits itself to return without doing anything.

**See also:**

**Find, First, Jump, Last, Previous, Row**

**OnLoad**

**Goal:**

To translate a binary buffer into a *storing object*.

**Syntax:**

public override void OnLoad(BinaryReader rd){

.

.

.

}

**Remarks:**

You must provide one **OnLoad** method for each *storing class* you define. You must use the **BinaryReader** provided to you by the system, to read the binary buffer and then populate the *storing object*'s data members. Your method must take care to read the data members in exactly the same order they were written by **OnSave**. String data members must be read on a fixed size base, so you need to call **ReadFix**, It is recommended to **Trim** any read string.

**See also:**

**TableFileName, OnSave, Path**

**OnSave**

**Goal:**

To translates a *storing object* to a sequence of bytes.

**Syntax:**

public override void OnSave(BinaryWriter wr){

.

.

.

}

**Remarks:**

You must provide one **OnSave** method for each *storing class* you define. You must use the **BinaryWriter** provided to you by the system, to write the binary buffer with the storing object's data members. Your method must take care to write the data members in exactly the same order they will be read by **OnLoad**. String data members must be written on a fixed size basis, to do so you need to call method **WriteFix**.

**See also:**

**Fix, TableFileName, OnLoad, Path**

**Open**

**Goal:**

To create a *view* for the specified table.

**Syntax:**

public void Open();

public void Open(mpIndexKey getKey);

public void Open(mpIndexKey getKey,string xName);

**Remarks:**

Creates a view for the table specified in **TableFileName**, if the table is closed it is open first, if **this** is bound to a previously created *view*, the previously created *view* is destroyed first. If no parameters are provided, the *view* bound to this is a *physical view*, a physical view allows access to all the rows contained on the table, in the sequence in which they were added to the table. If parameter ***getKey*** is specified, a *sorted view* is bound to **this**. A *sorted view* allows the *view* to see the table in an ordered way, but also to quickly find rows by key. To do so, the **mpIndexKey** method specified in ***getKey*** is executed once for each row on the table in order to assign a key to every row, keys are stored on a an index file which then is updated every time a change that affects the key occurs. If ***xName*** is not provided, the sort is assumed to be temporary and the index file is destroyed together with the view, otherwise the sort is considered permanent and ***xName*** is used as the name for permanent index file on disk, in this case when the view is destroyed the index file is closed instead of destroyed, so next time it is requested it can quickly be open instead of created over again from scratch. To make a permanent sort efficient, it must be created when the table is open and discarded only when no more updates to the table are possible, otherwise the next time you request such sort, the system will realize the table file was updated when the index file was closed and will force a full rebuild. This also happens when the view is not explicitly destroyed, i.e. when the application ends without destroying the *view* first.

The file name is assumed to be specified in **Name**, to have extension ".Table" and to exist at the default folder, which if not changed, is the same where the executable file resides. A call to static method **SetPath** can change the default path if necessary. To know what's the default use static method **GetPath**. Index files have extension ".Index".

An exception is thrown if the table doesn't exist on the default path, or when the row size doesn't match the current table class.

If you are using **gBase** on a **Unity** project it is highly recommended that you set the default path to **Application.persistentDataPath** before any database operation takes place.

**See also:**

**Create, Close, Destroy, Exists, GetPath, SetPath**

**Pack**

**Goal:**

To schedule a pack on the table.

**Syntax:**

public void Pack();

**Remarks:**

Schedules the physical deletion process to be executed when the table is closed. The process generates a backup file of the table before proceeding, then all rows flagged for deletions are removed from the table. The deletion process forces all permanent sorts to be out of synchrony and thus to be rebuild next time they are requested.

**See also:**

**Close, Delete, Deleted, Recall**

**RelativeRow**

**Goal:**

To retrieve the relative position of the row pointer.

**Syntax:**

public long RelativeRow();

**Remarks:**

Retrieves the relative position of the row pointer based on the sort order, the returned value is equivalent to the number of rows you would have to jump from the first row to reach the current row. This is useful when you display data on screen and need to calculate the slider’s position of the vertical scroll bar. Observe that if the current view has a physical sort, **RelativeRow** returns the same value than **Row**.

**See also:**

**Find, First, Jump, Last, Next, Row**

**Previous**

**Goal:**

To jump one row towards the top of the table (first row).

**Syntax:**

public void Previous();

**Remarks:**

If the row pointed by the *view* is not the first one, the *view* jumps one row towards the first row, and the newly pointed row is loaded into **this**. If the *view* is pointing to the first row, the *view* is reset to row zero and **this** is blanked.

**Previous** requires a valid row as starting point for the jump, when the *view* points to row zero no jump is possible, thus **Previous** limits itself to return without doing anything.

**See also:**

**Find, First, Jump, Last, Next, Row**

**Recall**

**Goal:**

To remove the physical deletion flag from a row.

**Syntax:**

public void Recall();

**Remarks:**

Removes the deletion flag from the row pointed by the *view*. No action is carried out if the row was not previously locked. To make the change permanent, please call **WriteAndUnlock**.

**See also:**

**Delete, Deleted, Lock, Pack, Unlock, WriteAndUnlock**

**Rows**

**Goal:**

To report number of rows contained on the *view*.

**Syntax:**

public long Rows();

**Remarks:**

Reports the number of rows contained on the *view*. If the *view* is filtered this number should be less or equal than the number of rows contained on the table. If the view has no filter this number is always equal to the number of rows stored on the table.

**See also:**

**First, Jump, Last, Row**

**Unlock**

**Goal:**

To releases a row lock.

**Syntax:**

public void Unlock();

**Remarks:**

If the lock count for the row pointed by the *view* is greater than one, it only decrements the lock count of that row by one and returns, otherwise it actually releases the lock, discarding any changes made to the data members. If you want any changes to become permanent you must call **WriteAndUnlock**. Rows must be unlocked at the soonest possible; otherwise you may be forcing other threads trying to lock the same row to be suspended. Threads must take care to call either **Unlock** or **WriteAndUnlock** once for every call made to **Lock**, otherwise the lock won't be released.

Although it is possible to leave the current row locked, jump to other row, come back to the original row and then unlock it, it is considered a bad practice and should be avoided.

An exception is thrown if the row pointed by the view is not locked.

**See also:**

**Add, Lock, WriteAndUnlock**

**WriteAndUnlock**

**Goal:**

To releases a row lock saving the data members to disk if necessary.

**Syntax:**

public void WriteAndUnlock();

**Remarks:**

If the lock count for the row pointed by the *view* is greater than one, it only decrements the lock count of that row by one and returns, otherwise the contents of the row on disk is replaced with the contents of **this**, if you want to unlock the row without writing it to disk you must call **Unlock**. Rows must be unlocked at the soonest possible; otherwise you may be forcing other threads trying to lock the same row to be suspended. Threads must take care to call either **Unlock** or **WriteAndUnlock** once for every call made to **Lock**, otherwise the lock won' be released and no file update will occur.

Although it is possible to leave the current row locked, jump to other row, come back to the original row and then unlock it, it is considered a bad practice and should be avoided.

An exception is thrown if the row pointed by the view is not locked.

**See also:**

**Add, Lock, Unlock**

**Variables**

**Name**

**Goal:**

To get or set the file name of the table.

**Syntax:**

public string Name;

**Remarks:**

Reports the name of the file in which the table is stored. It must be set before the table is open or created, any change made after the table is open has no effect.

**See also:**

**Create, Destroy, Exits, Open**

**Row**

**Goal:**

To get or set the physical row number.

**Syntax:**

public long Row;

**Remarks:**

Reports the physical row number corresponding to the row pointed by the *view*. When set to a new value, it moves the *view pointer* to the specified physical row number, the physical row number indicates the order in which the rows were added to the table. If the specified row exists and is visible to the current *view*, such row is loaded into **this**. If the destination row does not exist or is invisible to the current *view*, the *view* is reset to row zero, and **this** is blanked.

**See also:**

**Find, Jump, Last, Next, Previous, Rows**

**Static Methods**

**Fix**

**Goal:**

To generate a string with a fixed number of characters.

**Syntax:**

public static string Fix(string str,int size);

**Remarks:**

If the string in ***str*** is smaller than ***size*** characters then it pads as many tailing spaces as necessary to make the string length equal to ***size***. If the string length is greater than ***size***, then any extra characters are removed. The resulting string is guaranteed to contain exactly ***size*** characters.

**See also:**

**Find, OnSave**

**GetPath**

**Goal:**

To retrieve the default path.

**Syntax:**

public static string GetPath();

**Remarks:**

Retrieves the default path, which is the folder where all the files are assumed to exist.

**See also:**

**Create, Exists, Destroy, Open, SetPath**

**ReadFix**

**Goal:**

To read a string from the file buffer.

**Syntax:**

public static string ReadFix(BinaryReader rd,int size);

**Remarks:**

Reads a string from the file buffer, the string is assumed to be next object ready to be read on the file buffer, and to contain exactly ***size*** bytes. To prevent data loss, objects on the file buffer must be read exactly in the same order they were written, strings sizes must also match to those written. The returned string is trimmed, by removing all the spaces before the first non space character and all spaces after the last non space character.

**See also:**

**Fix, WriteFix**

**ReadFixFull**

**Goal:**

To read a string from the file buffer.

**Syntax:**

public static string ReadFix(BinaryReader rd,int size);

**Remarks:**

Reads a string from the file buffer, the string is assumed to be next object ready to be read on the file buffer, and to contain exactly ***size*** bytes. To prevent data loss, objects on the file buffer must be read exactly in the same order they were written, strings sizes must also match to those written. The returned string contains exactly ***size*** characters.

**See also:**

**Fix, WriteFix**

**SetPath**

**Goal:**

To change the default path.

**Syntax:**

public static void SetPath(string newPath);

**Remarks:**

Changes the default path to whatever is specified on ***newPath***. The programmer is responsible for making sure the name specified in ***newPath*** is valid in the current operating system context.

If you are using **gBase** on a **Unity** project it is highly recommended that you set the default path to **Application.persistentDataPath** before any database operation takes place.

**See also:**

**Create, Destroy, Exists, GetPath, Open**

**WriteFix**

**Goal:**

To write a string to the file buffer.

**Syntax:**

public static string WriteFix(BinaryWriter wr,string str,int size);

**Remarks:**

Writes a string of the specified to the file buffer. To prevent data lose, make sure the size specified in **WriteFix** is the exactly the same as the corresponding **ReadFix**. If the string member is used to generate index keys, make sure call static method fix to ensure fixed size keys.

**See also:**

**Fix, WriteFix**